# DeFi Protocol Suite / Privacy Mixer / AI Network Infrastructure

Audit Date: 2026-02-14 | Commit: 8a3f2c1 | Auditor: Claude Opus 4.6 | Contracts Reviewed: 47 | Lines of Code: 12,847

**Disclaimer:** This document is an AI-generated tool-assisted smart contract security audit report. This professional-quality report, conducted by Claude Opus 4.6, is not a substitute for a human audit. It may contain inaccuracies and omissions. Smart contracts are complex and software vulnerabilities are always a possibility. Use this report as one input among many, not as the sole basis for security decisions.

## Contract Overview

Overview

This audit covers 47 Solidity smart contracts for the DeFi protocol suite, implementing a privacy-preserving transaction (Tornado) mixer, delegated staking, governance, and a decentralized AI inference network. The protocol uses Groth16 zero-knowledge proofs for private withdrawals, EIP-712 typed signatures for gasless relayed transactions, and a custom staking reward distribution system with epoch-based checkpoints and slashing.

Protocol Suite

- Privacy Mixer (Tornado-style)
- Staking Pool
- Governance
- AI Network Staking
- Gasless Relayer

Contracts Reviewed

Mixer.sol · MerkleTree.sol · Verifier.sol · StakingRewards.sol · GovToken.sol · Governor.sol · Relayer.sol · AIStaking.sol · AIRewardDistributor.sol · ProofVerifier.sol · MerkleUtils.sol · EIP712Helper.sol

Contract Dependencies

OpenZeppelin v5.0 · Groth16 Verifier · Poseidon Hash · Custom Merkle Tree · Chainlink Price Feed · Gelato Relay · EIP-712 Helper · Custom Math Libraries

View full tree

Severity Distribution

- Critical 0
- High 3
- Medium 4
- Low 3
- Gas 2
- Informational 4

## High Severity 3

### 1. Unrestricted relayer drainage through executeCalls function

[critical] [fund_loss] [access_control] [relayer] [drainage]

The `executeCalls()` function in `Relayer.sol` allows any caller to execute arbitrary calls to any address with any calldata, without any validation, allowlist, or access control. Since the relayer holds ETH for gas reimbursement, an attacker can drain the relayer's entire balance by deploying a contract and calling `executeCalls()` with a single call to `msg.sender`.

Affected Function

```
function executeCalls(
    address[] calldata targets,
    bytes[] calldata calldatas
) external payable {
    for (uint256 i = 0; i < targets.length;
        (bool success, bytes memory ret) = targets[i].call{value: msg.value}(calldatas[i]);
        emit CallExecuted(targets[i], success, ret);
    }
}
```

Impact

An attacker can call `Relayer.executeCalls([victim], [0x], 0)` to drain the relayer's entire ETH balance, or craft calldata to call `transfer()` on any ERC20 token held by the relayer, or call any function on any contract to exploit third-party approvals granted to the relayer.

Fix

Add an allowlist of authorized targets and restrict calldata to known safe selectors. The relayer should only forward transactions to contracts it intends to support, not act as a general-purpose execution proxy for anyone.

```
// SPDX-License-Identifier: MIT
pragma Relayer.sol
contract Relayer {
    // ...
    mapping(address => bool) public isAllowedTarget;
    mapping(address => mapping(bytes4 => bool)) public isAllowedSelector;
    address public immutable owner;
    modifier onlyOwner() {
        require(msg.sender == owner, "not owner");
        _;
    }
    function setAllowedTarget(address target, bool allowed) external onlyOwner {
        isAllowedTarget[target] = allowed;
        emit AllowedTargetSet(target, allowed);
    }
    function setAllowedSelector(address target, bytes4 selector, bool allowed) external onlyOwner {
        isAllowedSelector[target][selector] = allowed;
        emit AllowedSelectorSet(target, selector, allowed);
    }
    function executeCalls(address[] calldata targets, bytes[] calldata calldatas) external payable {
        for (uint256 i = 0; i < targets.length; i++) {
            address t = targets[i];
            bytes4 sel = bytes4(calldatas[i][0:4]);
            require(isAllowedTarget[t], "target not allowed");
            require(isAllowedSelector[t][sel], "selector not allowed");
            (bool success, bytes memory ret) = t.call{value: msg.value}(calldatas[i]);
            emit CallExecuted(t, success, ret);
        }
    }
}
```

### 2. Denial of Service through unbounded loop in weight calculation

[critical] [dos] [gas] [governance] [staking] [weight] [checkpoint]

The governance and staking reward systems iterate over all historical checkpoints to find the latest one. As checkpoints accumulate, this loop will exceed the block gas limit, permanently bricking voting and reward claims. Voting power is used for governance proposals and staking rewards are distributed proportionally. Once the checkpoint count grows beyond ~300, no new votes can be cast and no rewards can be claimed.

Affected Function

```
function getPastVotes(address account, uint256 blockNumber) public view {
    uint32 nCheckpoints = numCheckpoints[account];
    for (uint32 i = nCheckpoints - 1; i >= 0; i--) {
        if (checkpoints[account][i].fromBlock <= blockNumber) {
            return checkpoints[account][i].votes;
        }
    }
    return 0;
}
```

Impact

Governance becomes permanently unusable once the checkpoint count exceeds the gas limit. Stakers lose their accumulated rewards. Tokens become locked in governance contracts with no way to recover them.

Fix

Use binary search instead of linear iteration. Checkpoints are sorted by block number, so binary search finds the correct checkpoint in O(log n) instead of O(n).

```
function getPastVotes(address account, uint256 blockNumber) public view {
    uint32 nCheckpoints = numCheckpoints[account];
    if (nCheckpoints == 0) return 0;
    if (checkpoints[account][0].fromBlock > blockNumber) return 0;
    if (checkpoints[account][nCheckpoints - 1].fromBlock <= blockNumber) {
        return checkpoints[account][nCheckpoints - 1].votes;
    }
    uint32 lower = 0;
    uint32 upper = nCheckpoints - 1;
    while (upper > lower) {
        uint32 mid = upper - (upper - lower) / 2;
        Checkpoint memory cp = checkpoints[account][mid];
        if (cp.fromBlock == blockNumber) {
            return cp.votes;
        } else if (cp.fromBlock < blockNumber) {
            lower = mid;
        } else {
            upper = mid - 1;
        }
    }
    return checkpoints[account][lower].votes;
}
```

### 3. Proof forgery through unbounded loop in merkle verification

[critical] [proof_forgery] [merkle_tree] [verifier] [loop]

The `MerkleUtils.verifyProof()` function iterates over the proof array without validating its length. Since the Merkle tree has a fixed depth of 20 levels, an attacker can provide a proof longer than 20 elements to manipulate the hashing chain and forge a valid proof for any leaf that was never deposited into the mixer.

Affected Function

```
function verifyProof(
    bytes32[] calldata proof,
    bytes32 root,
    bytes32 leaf
) internal pure {
    bytes32 computedHash = leaf;
    for (uint256 i = 0; i < proof.length; i++) {
        bytes32 sibling = proof[i];
        if (computedHash <= sibling) {
            computedHash = keccak256(abi.encodePacked(computedHash, sibling));
        } else {
            computedHash = keccak256(abi.encodePacked(sibling, computedHash));
        }
    }
    require(computedHash == root, "invalid proof");
}
```

Impact

An attacker can submit a proof longer than 20 elements to manipulate the hashing chain and forge a valid proof for any leaf that was never deposited into the mixer. This allows withdrawing funds without a legitimate deposit.

Fix

Add a length check at the beginning of the function: `require(proof.length == 20, "invalid proof length");`

## Medium Severity 4

### 1. Reentrancy vulnerability in ERC20Transfer withdrawal with ETH refunds

[medium] [reentrancy] [erc20] [withdrawal] [refund] [eth]

The `_processWithdrawal()` function in `Mixer.sol` processes ERC20 withdrawals and refunds any excess ETH sent with the transaction. The refund is sent before the withdrawal is marked as complete. If the recipient is a contract, it can reenter `withdraw()` with the same proof and nullifier before the first call completes, bypassing the nullifier check and draining additional tokens.

Affected Function

```
function _processWithdrawal(
    address token,
    address payable recipient,
    uint256 amount,
    uint256 relayerFee,
    bytes32 nullifierHash
) internal {
    // ERC20 transfer
    IERC20(token).safeTransfer(recipient, amount - relayerFee);

    // ETH refund if msg.value > 0
    if (msg.value > 0) {
        uint256 refund = msg.value - relayerFee;
        if (refund > 0) {
            (bool success, ) = msg.sender.call{value: refund}("");
            require(success, "refund failed");
        }
    }

    // Mark nullifier as spent
    nullifierHashes[nullifierHash] = true;
}
```

Impact

An attacker can drain tokens from the mixer by reentering the withdrawal function before the nullifier is marked as spent. The attacker receives more tokens than their legitimate deposit entitles them to, at the expense of other depositors.

Fix

Follow the checks-effects-interactions pattern: mark the nullifier as spent before any external calls. Alternatively, add a reentrancy guard.

```
function _processWithdrawal(
    address token,
    address payable recipient,
    uint256 amount,
    uint256 relayerFee,
    bytes32 nullifierHash
) internal {
    // Mark nullifier as spent FIRST
    nullifierHashes[nullifierHash] = true;

    // ERC20 transfer
    IERC20(token).safeTransfer(recipient, amount - relayerFee);

    // ETH refund if msg.value > 0
    if (msg.value > 0) {
        uint256 refund = msg.value - relayerFee;
        if (refund > 0) {
            (bool success, ) = msg.sender.call{value: refund}("");
            require(success, "refund failed");
        }
    }
}
```

### 2. Fee-sniping vulnerability in deposit accumulation mechanism

[medium] [fee_sniping] [deposit] [accumulation] [mev]

The `depositAndCommit()` function in `FeeSniping.sol` allows anyone to deposit into the mixer and immediately claim the accumulated fee. This creates a MEV opportunity: a bot can monitor the mempool for large deposits, front-run them with a small deposit, and collect the accumulated fee. The attacker's deposit is then processed like any other deposit, diluting the fee pool for subsequent depositors.

Affected Function

```
function depositAndCommit(
    bytes32 commitment,
    uint256 feeAmount,
    address payable feeRecipient
) external {
    // ...
    if (feeAmount > 0) {
        (bool success, ) = feeRecipient.call{value: feeAmount}("");
        require(success, "fee transfer failed");
    }
    // ...
}
```

Impact

An attacker can monitor the mempool for large deposits, front-run them with a small deposit, and immediately claim the accumulated fee. The attacker's deposit is then processed like any other deposit, diluting the fee pool for subsequent depositors.

Fix

Implement a time-delayed fee accumulation mechanism. Instead of immediate payout, accumulate fees in a pool and allow claims only after a minimum number of blocks have passed. This makes the attack unprofitable because the attacker must wait while other deposits dilute the fee pool.

### 3. Integer overflow in weight normalization calculation

[medium] [overflow] [weight] [normalization] [staking]

The `getNormalizedWeight()` function in `StakingRewards.sol` multiplies a user's staked amount by a fixed 1e18 precision factor before dividing by the total stake. For large stakes, the multiplication overflows the uint256 limit, causing incorrect weight calculations. This affects reward distribution and governance voting power.

Affected Function

```
function getNormalizedWeight(
    address user
) public view {
    uint256 userStake = stakes[user].amount;
    uint256 totalStake = totalStaked;
    return (userStake * 1e18) / totalStake;
}
```

Impact

Overflow causes incorrect weight calculations, leading to rewards being distributed proportional to a fraction of the intended. The bounds for implementation are not specified.

Fix

Use a safe multiplication library or restrict the maximum stake to prevent overflow.

### 4. Unchecked external call success in Curvefinance gas refund

[medium] [unchecked_call] [gas_refund] [curvefinance] [silent_failure]

The `_refundGas()` function in `GasRefund.sol` calls `tx.origin.call{value: refunds[tx.origin]}("")` to refund accumulated gas fees, but never checks the return value. If the call fails (e.g. because the recipient is a contract that rejects ETH), the refund is silently lost. The contract still marks the refund as processed, permanently locking the ETH in the contract.

Affected Function

```
function _refundGas(
    address[] calldata targets,
    bytes[] calldata calldatas
) internal {
    uint256 startGas = gasleft();
    // ...
    uint256 gasUsed = startGas - gasleft();
    uint256 refund = gasUsed * tx.gasprice;
    if (refund > address(this).balance) {
        refund = address(this).balance;
    }
    refunds[tx.origin] += refund;
    tx.origin.call{value: refund}("");
    emit GasRefunded(tx.origin, refund);
}
```

Impact

If refund calls keep failing, the contract's balance slowly drains as the accumulated refunds are never cleared. The attacker can repeatedly trigger failed refunds to drain the contract's ETH balance.

Fix

Check the return value and only clear the refund if the call succeeds.

## Low Severity 3

### 1. Missing validation of zero denomination in Tornado contractor

[low] [validation] [zero_denomination] [constructor] [deployment]

The `Mixer.sol` constructor does not validate that the denomination parameter is non-zero. A deployment mistake would create a mixer where every deposit and withdrawal requires zero ETH, rendering the contract useless.

Affected Function

```
contract Mixer {
    IVerifier public verifier;
    IHasher public hasher;
    uint256 public denomination;
    uint32 public merkleTreeHeight;
    uint32 public constant ROOT_HISTORY_SIZE = 30;
    uint32 public currentRootIndex = 0;
    uint32 public nextIndex = 0;
    mapping(bytes32 => bool) public nullifierHashes;
    mapping(bytes32 => bool) public commitments;
    mapping(uint32 => bytes32) public roots;

    constructor(
        IVerifier _verifier,
        IHasher _hasher,
        uint256 _denomination,
        uint32 _merkleTreeHeight
    ) {
        verifier = _verifier;
        hasher = _hasher;
        denomination = _denomination;
        merkleTreeHeight = _merkleTreeHeight;
        emit MerkleTreeInit(_merkleTreeHeight, _denomination);
    }
}
```

Impact

Deploying a mixer with zero denomination creates a mixer where every deposit and withdrawal requires zero ETH, rendering the contract useless.

Fix

Add a `require(_denomination > 0, "denomination must be > 0");` check in the constructor.

### 2. Missing event emission for critical state changes

[low] [events] [monitoring] [governance] [staking]

The `GovToken.sol` and `StakingRewards.sol` contracts modify critical state but doesn't emit any events, making it difficult to track proposal creation, vote casting, stake changes, and reward distribution off-chain.

Affected Function

```
function castVote(uint256 proposalId, uint8 support) external {
    // ...
    hasVoted[proposalId][msg.sender] = true;
    // ...
}
```

Impact

Off-chain indexers, dashboards, and governance tooling cannot reliably track protocol activity. Users have no easy way to monitor their own stakes, votes, or rewards without scanning raw storage slots.

Fix

Add events for all state changes: `ProposalCreated`, `VoteCast`, `Staked`, `Unstaked`, `RewardClaimed`, `RewardDistributed`.

### 3. Function body transition due to parameters setting

[low] [function_body] [transition] [parameters] [governance]

The governance contract allows proposals to change function implementations. A malicious proposal could permanently bricking protocol functionality. Timelock delays give users time to exit before changes take effect, but the underlying vulnerability remains.

Affected Function

```
function execute(uint256 proposalId) external {
    // ...
    (bool success, ) = target.call(calldata);
    // ...
}
```

Impact

Governance can change the implementation of any function, potentially bricking the protocol or stealing funds. Timelock delays give users time to exit before changes take effect, but the underlying vulnerability remains.

Fix

Restrict which contracts governance can target, or implement a whitelist of allowed function selectors.

## Informational 4

### 1. Inconsistent EIP712 domain function creates accounting risks

[info] [eip712] [domain] [inconsistent] [accounting]

The `EIP712Helper.sol` contract implements EIP712 typed data hashing but uses different parameters for the domain separator than the main `Relayer.sol` contract. This creates a risk that signatures generated for one contract could be replayed against the other, or that users will sign messages they don't intend to sign because the wallet shows a different domain than expected.

Affected Function

```
function _domainSeparatorV4() internal view {
    return keccak256(abi.encode(
        _TYPE_HASH,
        keccak256(bytes("Relayer")),
        keccak256(bytes("1")),
        block.chainid,
        address(this)
    ));
}
```

Impact

Signatures intended for one contract may be valid for the other, or users may sign messages they don't intend to sign because the wallet shows a different domain than expected.

Fix

Use a single, shared EIP712 domain for all contracts in the protocol. Define the domain once in a library and import it everywhere.

### 2. Function body transition due to parameters setting

[info] [function_body] [transition] [parameters] [governance]

The governance contract allows proposals to change function implementations. A malicious proposal could permanently bricking protocol functionality. Timelock delays give users time to exit before changes take effect, but the underlying vulnerability remains.

Affected Function

```
function execute(uint256 proposalId) external {
    // ...
    (bool success, ) = target.call(calldata);
    // ...
}
```

Impact

Governance can change the implementation of any function, potentially bricking the protocol or stealing funds. Timelock delays give users time to exit before changes take effect, but the underlying vulnerability remains.

Fix

Restrict which contracts governance can target, or implement a whitelist of allowed function selectors.

### 3. Gas optimization: Redundant storage reads in deposit tracker

[info] [gas] [storage] [deposit] [optimization]

The `depositTracker.sol` contract performs multiple storage reads for the same value, wasting gas. The `getDepositCount()` function reads `totalDeposits` twice in two lines of the `recordDeposit()` function.

Affected Function

```
function recordDeposit(address depositor, uint256 amount) {
    uint256 count = totalDeposits;
    emit DepositRecorded(depositor, amount, totalDeposits);
    totalDeposits = count + 1;
}
```

Impact

Wasted gas on unnecessary storage reads. Low impact on mainnet but adds up over many transactions.

Fix

Cache the storage read in a local variable: `uint256 count = totalDeposits;`

### 4. Gas optimization: Redundant storage reads in deposit tracker

[info] [gas] [storage] [deposit] [optimization]

The `depositTracker.sol` contract performs multiple storage reads for the same value, wasting gas. The `getLastDepositTime()` function reads `lastDeposit[msg.sender]` twice in two lines of the `recordDeposit()` function.

Affected Function

```
function recordDeposit(address depositor, uint256 amount) {
    uint256 time = lastDeposit[msg.sender];
    emit DepositRecorded(depositor, amount, lastDeposit[msg.sender]);
    lastDeposit[msg.sender] = block.timestamp;
}
```

Impact

Wasted gas on unnecessary storage reads. Low impact on mainnet but adds up over many transactions.

Fix

Cache the storage read in a local variable: `uint256 time = lastDeposit[msg.sender];`


## Need a Professional Security Audit?

While AI provides quick insights, professional auditors catch what AI misses.

**Cantrips Credentials**

- Hundreds of DeFi security audits completed
- Building some of the world's leading protocols
- Comprehensive and thoroughly documented reports

**Professional Benefits**

- Manual code review by security experts
- Extended vulnerability guarantees
- Formal verification reports
- Ongoing security partnership

See Professional Audit ↗ · Contact Sales ↗